Университет ИТМО

Лабораторная работа №2

«Изучение алгоритмов поиска»

Системы искусственного интеллекта

Выполнил:

Бавыкин Роман Алексеевич

P33101

Вариант 8

г. Санкт-Петербург

2022

Задание:

Цель задания: Исследование алгоритмов решения задач методом поиска.

Описание предметной области. Имеется транспортная сеть, связывающая города СНГ. Сеть представлена в виде таблицы связей между городами. Связи являются двусторонними, т.е. допускают движение в обоих направлениях.

Необходимо проложить маршрут из одной заданной точки в другую.

Этап 1. Неинформированный поиск. На этом этапе известна только топология связей между городами. Выполнить:

1) поиск в ширину
2) поиск глубину;
3) поиск с ограничением глубины;
4) поиск с итеративным углублением;
5) двунаправленный поиск.

Отобразить движение по дереву на его графе с указанием сложности каждого вида поиска. Сделать выводы.

Этап 2. Информированный поиск. Воспользовавшись информацией о протяженности связей от текущего узла, выполнить:

1) жадный поиск по первому наилучшему соответствию;

2) затем, использую информацию о расстоянии до цели по прямой от каждого узла, выполнить поиск методом минимизации суммарной оценки A*.

Отобразить на графе выбранный маршрут и сравнить его сложность с неинформированным поиском. Сделать выводы.

Вариант: Вильнюс – Одесса

Решение:

Полный код доступен по ссылке: https://github.com/robqqq/ai

```
vilnius = "Вильнюс"
vitebsk = "Витебск"
brest = "Брест"
voronezh = "Воронеж"
volgograd = "Волгоград"
```

```
novgorod = "Нижний Новгород"
daugavpils = "Даугавпилс"
kaliningrad = "Калининград"
kaunas = "Каунас"
kiev = "Киев"
zhitomir = "Житомир"
donetsk = "Донецк"
kishinev = "Кишинев"
st_petersburg = "Санкт-Петербург"
riga = "Рига"
moscow = "Москва"
kazan = "Казань"
minsk = "Минск"
murmansk = "Мурманск"
orel = "Орел"
odessa = "Одесса"
tallinn = "Таллинн"
kharkiv = "Харьков"
symferopol = "Симферополь"
yaroslavl = "Ярославль"
ufa = "Уфа"
samara = "Самара"


graph = {
    vilnius: [brest, daugavpils, vitebsk, kaliningrad, kaunas, kiev],
    vitebsk: [brest, vilnius, novgorod, voronezh, volgograd, st_petersburg,
orel],
    brest: [vilnius, vitebsk, kaliningrad],
    voronezh: [vitebsk, volgograd, yaroslavl],
    volgograd: [vitebsk, voronezh, zhitomir],
    novgorod: [vitebsk, moscow],
    daugavpils: [vilnius],
    kaliningrad: [brest, vilnius, st_petersburg],
    kaunas: [vilnius, riga],
    kiev: [vilnius, zhitomir, kishinev, odessa, kharkiv],
    zhitomir: [donetsk, volgograd, kiev],
    donetsk: [zhitomir, kishinev, moscow, orel],
    kishinev: [kiev, donetsk],
    st_petersburg: [vitebsk, kaliningrad, riga, moscow, murmansk],
    riga: [kaunas, st_petersburg, tallinn],
    moscow: [kazan, novgorod, minsk, donetsk, st_petersburg, orel],
    kazan: [moscow, ufa],
    minsk: [moscow, murmansk, yaroslavl],
```

```
    murmansk: [st_petersburg, minsk],
    orel: [vitebsk, donetsk, moscow],
    odessa: [kiev],
    tallinn: [riga],
    kharkiv: [kiev, symferopol],
    symferopol: [kharkiv],
    yaroslavl: [voronezh, minsk],
    ufa: [kazan, samara],
    samara: [ufa]
}

distances: dict[tuple[str, str], int] = {
    (vilnius, brest): 531,
    (vitebsk, brest): 638,
    (vitebsk, vilnius): 360,
    (voronezh, vitebsk): 869,
    (voronezh, volgograd): 581,
    (volgograd, vitebsk): 1455,
    (vitebsk, novgorod): 911,
    (vilnius, daugavpils): 211,
    (kaliningrad, brest): 699,
    (kaliningrad, vilnius): 333,
    (kaunas, vilnius): 102,
    (kiev, vilnius): 734,
    (kiev, zhitomir): 131,
    (zhitomir, donetsk): 863,
    (zhitomir, volgograd): 1493,
    (kishinev, kiev): 467,
    (kishinev, donetsk): 812,
    (st_petersburg, vitebsk): 602,
    (st_petersburg, kaliningrad): 739,
    (st_petersburg, riga): 641,
    (moscow, kazan): 815,
    (moscow, novgorod): 411,
    (moscow, minsk): 690,
    (moscow, donetsk): 1084,
    (moscow, st_petersburg): 664,
    (murmansk, st_petersburg): 1412,
    (murmansk, minsk): 2238,
    (orel, vitebsk): 522,
    (orel, donetsk): 709,
    (orel, moscow): 368,
    (odessa, kiev): 487,
```

```
    (riga, kaunas): 267,
    (tallinn, riga): 308,
    (kharkiv, kiev): 471,
    (kharkiv, symferopol): 639,
    (yaroslavl, voronezh): 739,
    (yaroslavl, minsk): 940,
    (ufa, kazan): 525,
    (ufa, samara): 461
}

heuristics = {
    vilnius: 991,
    vitebsk: 968,
    brest: 806,
    voronezh: 844,
    volgograd: 1064,
    novgorod: 1427,
    daugavpils: 1083,
    kaliningrad: 1163,
    kaunas: 1052,
    kiev: 441,
    zhitomir: 447,
    donetsk: 563,
    kishinev: 155,
    st_petersburg: 1502,
    riga: 1249,
    moscow: 1138,
    kazan: 1643,
    minsk: 854,
    murmansk: 2507,
    orel: 816,
    odessa: 0,
    tallinn: 1494,
    kharkiv: 564,
    symferopol: 314,
    yaroslavl: 1386,
    ufa: 1993,
    samara: 1574
}

distances.update({(distance[1], distance[0]): distances[distance] for
distance in distances})
```

1) Поиск в ширину:

```
def breadth_first_search(graph: dict[str, list[str]], start: str, goal: str)
-> list[str]:

    routes = [[start]]
    new_routes = []
    if start == goal:
        return [start]
    while True:
        for route in routes:
            for town in graph[route[-1]]:
                if town == goal:
                    return route + [town]
                else:
                    new_routes += [route + [town]]
        routes = new_routes
        new_routes = []

print(breadth_first_search(graph, vilnius, odessa))
```

```
['Вильнюс', 'Киев', 'Одесса']
```

2) Поиск в глубину:

```
def depth_first_search(graph: dict[str, list[str]], start: str, goal: str) ->
list[str] | None:
    route = [start]
    visited = [start]
    if start == goal:
        return route
    while len(visited) != len(graph):
        next = False
        for town in graph[route[-1]]:
            if (town not in visited):
                route += [town]
                if town == goal:
                    return route
                visited += [town]
                next = True
                break
        if not next:
            route = route[:-1]

print(depth_first_search(graph, vilnius, odessa))
```

```
['Вильнюс', 'Брест', 'Витебск', 'Нижний Новгород', 'Москва', 'Минск',
'Ярославль', 'Воронеж', 'Волгоград', 'Житомир', 'Донецк', 'Кишинев', 'Киев',
'Одесса']
```

3) Поиск с ограничением глубины:

```
def depth_limited_search(graph: dict[str, list[str]], start: str, goal: str,
limit: int) -> list[str] | None:
    route = [start]
    visited = [start]
    depth = 0
    if start == goal:
        return route
    while len(visited) != len(graph):
        next = False
        for town in graph[route[-1]]:
            if (town not in visited):
                route += [town]
                if town == goal:
                    return route
                visited += [town]
                next = True
                depth += 1
                break
        if not next or depth > limit:
            depth -= 1
            route = route[:-1]

print(depth_limited_search(graph, vilnius, odessa, 3))
```

```
['Вильнюс', 'Киев', 'Одесса']
```

4) Поиск с итеративным углублением:

```
def iterative_deepening_depth_first_search(graph: dict[str, list[str]],
start: str, goal: str) -> list[str] | None:
    route = [start]
    visited = [start]
    depth = 0
    limit = 1
    if start == goal:
        return route
    while len(visited) != len(graph):
        next = False
        for town in graph[route[-1]]:
            if (town not in visited):
```

```
                route += [town]
                if town == goal:
                    return route
                visited += [town]
                next = True
                depth += 1
                break
        if (depth == 0 and not next):
            visited = [start]
            limit += 1
            depth = 0
        elif not next or depth > limit:
            depth -= 1
            route = route[:-1]

print(iterative_deepening_depth_first_search(graph, vilnius, odessa))
```

```
['Вильнюс', 'Киев', 'Одесса']
```

5) Двунаправленный поиск:

```
def bidirectional_search(graph: dict[str, list[str]], start: str, goal: str)
-> list[str] | None:
    route: dict[bool, list[str]] = {False: [start], True: [goal]}
    visited: dict[bool, list[str]] = {False: [start], True: [goal]}
    curr = False
    depth: dict[bool, int] = {False: 0, True: 0}
    limit: dict[bool, int] = {False: 1, True: 1}
    if start == goal:
        return route[curr]
    while len(visited[curr]) != len(graph):
        next = False
        for town in graph[route[curr][-1]]:
            if (town not in visited[curr]):
                if town == route[not curr][-1]:
                    return route[False] + route[True][::-1]
                route = {curr: route[curr] + [town], not curr: route[not
curr]}
                visited = {curr: visited[curr] + [town], not curr:
visited[not curr]}
                next = True
                depth = {curr: depth[curr] + 1, not curr: depth[not curr]}
                break
        if (depth[curr] == 0 and not next):
            visited = {curr: [route[curr][0]], not curr: visited[not curr]}
```

```
            limit = {curr: 1, not curr: limit[not curr]}
            depth = {curr: 0, not curr: depth[not curr]}
        elif not next or depth[curr] > limit[curr]:
            depth = {curr: depth[curr] - 1, not curr: depth[not curr]}
            route = {curr: route[curr][:-1], not curr: route[not curr]}
        curr = not curr

print(bidirectional_search(graph, vilnius, odessa))
```

```
['Вильнюс', 'Киев', 'Одесса']
```

6) Жадный поиск по первому наилучшему соответствию:

```
def best_first_cmp(route1: tuple[str, str], route2: tuple[str, str]) -> int:
    return distances[route1] - distances[route2]

def best_first_search(graph: dict[str, list[str]], start: str, goal: str):
    route: list[str] = [start]
    visited = [start]
    if start == goal:
        return (route, 0)
    while len(visited) != len(graph):
        next = False
        for town in sorted([(route[-1], x) for x in graph[route[-1]]],
key=cmp_to_key(best_first_cmp)):
            if (town[1] not in visited):
                route += [town[1]]
                if town[1] == goal:
                    distance = 0
                    for i in range(len(route) - 1):
                        distance += distances[(route[i], route[i + 1])]
                    return (route, distance)
                visited += [town[1]]
                next = True
                break
        if not next:
            route = route[:-1]

best_first_search_results = best_first_search(graph, vilnius, odessa)
print(best_first_search_results[0], '-', best_first_search_results[1])
```

```
['Вильнюс', 'Каунас', 'Рига', 'Санкт-Петербург', 'Витебск', 'Орел', 'Москва',
'Минск', 'Ярославль', 'Воронеж', 'Волгоград', 'Житомир', 'Киев', 'Одесса'] -
7563
```

7) Поиск методом минимизации суммарной оценки A*:

```
def min_overall_score_cmp(route1: tuple[str, str], route2: tuple[str, str])
-> int:
    return distances[route1] + heuristics[route1[1]] - (distances[route2] +
heuristics[route2[1]])

def min_overall_score(graph, start, goal):
    route: list[str] = [start]
    visited = [start]
    if start == goal:
        return (route, 0)
    while len(visited) != len(graph):
        next = False
        for town in sorted([(route[-1], x) for x in graph[route[-1]]],
key=cmp_to_key(min_overall_score_cmp)):
            if (town[1] not in visited):
                route += [town[1]]
                if town[1] == goal:
                    distance = 0
                    for i in range(len(route) - 1):
                        distance += distances[(route[i], route[i + 1])]
                    return (route, distance)
                visited += [town[1]]
                next = True
                break
        if not next:
            route = route[:-1]

min_overall_score_results = min_overall_score(graph, vilnius, odessa)
print(min_overall_score_results[0], '-', min_overall_score_results[1])
```

```
['Вильнюс', 'Каунас', 'Рига', 'Санкт-Петербург', 'Витебск', 'Орел', 'Донецк',
'Кишинев', 'Киев', 'Одесса'] - 4609
```